BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for ALBUBUX BEP20 Token Smart Contract

Request Date: 2025-09-19
Completion Date: 2025-09-20
Language: Solidity

Contents

Commission

Audited Project	ALBUBUX BEP20 Token Smart Contract
Smart Contract Address	0x4843588e66700eD5E2C4F8BC6f9b61E686d9cDa9
Contract Creator	0xeBfBF0F46ebeaf7a8706873F97348520bdBaD701
Contract Owner	0xeBfBF0F46ebeaf7a8706873F97348520bdBaD701
Blockchain Network	Binance Smart Chain Mainnet

Block Solutions was commissioned by ALBUBUX BEP20 Token Smart Contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

ALBUBUX Properties

Contract Token name	Albubux
Total supply	4,000,000,000,000,000 BUBX
Symbol	BUBX
Decimals	9
Holders	6
Total Transfers	23
Buy Fee	1.02% (same as general transfer unless the pair or buyer is excluded).
Sell Fee	1.02% (same as general transfer unless excluded).
Transfer Fee	1.02% (applies unless sender or recipient is excluded).
Charity Tax	1 %
Auto Liquidity	0.01%
Reflection (redistributed to holders)	0.01%
Charity Address	0x271AA85AA4CD2e4CE45D97823fb0eeeb8334c908
Uniswap V2 Router	0x10ED43C718714eb63d5aA57B78B54704E256024E
Uniswap V2 Pair	0x6d9C130A9D7ea7078D16aD35D6907f034487483C
Smart Contract Address	0x4843588e66700eD5E2C4F8BC6f9b61E686d9cDa9
Contract Creator	0xeBfBF0F46ebeaf7a8706873F97348520bdBaD701
Contract Owner	0xeBfBF0F46ebeaf7a8706873F97348520bdBaD701
Blockchain Network	Binance Smart Chain Mainnet

Contract Functions

Executables

i. function approve(address spender, uint256 amount)public override returns (bool)
ii. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
iii. function deliver(uint256 tAmount) public
iv. function excludeFromReward(address account) public onlyOwner
v. function excludeFromFee(address account) public onlyOwner
vi. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
vii. function renounceOwnership() public virtual onlyOwner
viii. function setCharityFeePercent(uint256 charityFeeBps) external onlyOwner
ix. function setLiquidityFeePercent(uint256 liquidityFeeBps) external onlyOwner
x. function setSwapBackSettings(uint256 _amount) external onlyOwner
xi. function setTaxFeePercent(uint256 taxFeeBps) external onlyOwner
xii. function transferOwnership(address newOwner) public virtual onlyOwner
xiii. function transfer(address recipient, uint256 amount) public override returns (bool)
xiv. function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool)
xv. function includeInReward(address account) external onlyOwner

Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with block gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed
Fallback function security.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed

Whitepaper-Website-Contract correlation.	Passed
Front Running.	Passed

Executable Functions

ALBUBUX BEP20 TOKEN Contract

This will transfer tokens from the caller's wallet to the specified recipient address. The amount is the number of tokens to be transferred. The transfer() function overrides the standard ERC-20 logic and calls the internal _transfer function, which applies this token's custom rules such as transaction tax, reflection distribution, liquidity collection, and charity allocation before completing the transfer.

```
function transfer(address recipient, uint256 amount) public
override returns (bool)
{
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Leaves the contract without owner. It will not be possible to call `onlyOwner` functions anymore. Can only be called by the current owner. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner

```
function renounceOwnership() public virtual onlyOwner {
    _setOwner(address(0));
}
```

This will let the contract owner update the reflection tax percentage, where taxFeeBps is given in basis points (e.g., 100 = 1%). The function updates _taxFee and then checks that the sum of all fees (_taxFee + _liquidityFee + _charityFee) does not exceed the maximum limit of 25% (MAX_FEE).

```
function setTaxFeePercent(uint256 taxFeeBps) external onlyOwner {
    _taxFee = taxFeeBps;
    require(
        _taxFee + _liquidityFee + _charityFee <= MAX_FEE,
        "Total fee is over 25%"
    );
}
```

This will transfer tokens from a specified sender address to a recipient address, using the allowance previously approved by the sender. The amount is the number of tokens to be transferred. The transferFrom() function overrides the standard ERC-20 logic by calling the internal _transfer function, which applies this token's custom rules such as transaction tax, reflection distribution, liquidity collection, and charity allocation. After the transfer, it decreases the spender's allowance by the transferred amount, ensuring the spender cannot exceed the approved limit.

```
function transferFrom( address sender, address recipient, uint256 amount)
public override returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(
        sender,
        _msgSender(),
        _allowances[sender][_msgSender()].sub(
            amount,
            "ERC20: transfer amount exceeds allowance"
        )
    );
    return true;
}
```

Approve the passed address to spend the specified number of tokens on behalf of msg. sender. "spender" is the address which will spend the funds. "amount" the number of tokens to be spent.

```
function approve(address spender, uint256 amount)public override returns (bool)
{
    _approve(_msgSender(), spender, amount);
    return true;
}
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}
```

The includeInRewards() function allows the contract owner to re-include addresses in the reflection (rewards) mechanism, enabling it to receive proportional token rewards again.

```
function includeInReward(address account) external onlyOwner {
    require(_isExcluded[account], "Account is already excluded");
    for (uint256 i = 0; i < _excluded.length; i++) {
        if (_excluded[i] == account) {
            _excluded[i] = _excluded[_excluded.length - 1];
            _tOwned[account] = 0;
            _isExcluded[account] = false;
            _excluded.pop();
            break;
        }
    }
}
```

The excludeFromRewards() function allows the owner to exclude a specific address from receiving reflection (reward) distributions by calling the parent _isExluded() method.

```
function excludeFromReward(address account) public onlyOwner {
    // require(account != 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D,
    // 'We can not exclude Uniswap router.');
    require(!_isExcluded[account], "Account is already excluded");
    if (_rOwned[account] > 0) {
        _tOwned[account] = tokenFromReflection(_rOwned[account]);
    }
    _isExcluded[account] = true;
    _excluded.push(account);
}
```

This function allows the owner to exclude an address from transaction fees and transfer limits, enforcing a cap on the total number of such exclusions to prevent abuse.

```
function excludeFromFee(address account) public onlyOwner {
    _isExcludedFromFee[account] = true;
}
```

This function increases the allowance of a spender by a specified addedValue for the caller (owner), enabling the spender to spend more tokens on the owner's behalf.

```
function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool)
{
    _approve(
        _msgSender(),
        spender,
        _allowances[_msgSender()][spender].add(addedValue)
    );
    return true;
}
```

This function decreases the allowance granted to a spender by subtractedValue for the caller (owner), ensuring the allowance doesn't go below zero. It uses unchecked to optimize gas once the safety check passes.

```
function decreaseAllowance(address spender, uint256 subtractedValue) public
 virtual returns (bool)
{
    _approve(
        _msgSender(),
        spender,
        _allowances[_msgSender()][spender].sub(
            subtractedValue,
            "ERC20: decreased allowance below zero"
        )
    );
    return true;
}
```

This will let the contract owner update the liquidity fee percentage, where liquidityFeeBps is given in basis points (e.g., 100 = 1%). The function updates _liquidityFee and then checks that the combined total of all fees (_taxFee + _liquidityFee + _charityFee) does not exceed the maximum cap of 25% (MAX_FEE).

```
function setLiquidityFeePercent(uint256 liquidityFeeBps) external onlyOwner
{
    _liquidityFee = liquidityFeeBps;
    require(
        _taxFee + _liquidityFee + _charityFee <= MAX_FEE,
        "Total fee is over 25%"
    );
}
```

This will allow a non-excluded holder to "give up" (or deliver) a specified token amount tAmount back into the reflection pool. The function first checks that the caller (sender) is not excluded from rewards, because excluded accounts can't interact with reflection. It then calculates the reflected value rAmount for the given tAmount using _getValues(), subtracts that from the sender's reflected balance _rOwned[sender], reduces the global reflection supply _rTotal, and increases the total fees counter _tFeeTotal.

```
function deliver(uint256 tAmount) public {
    address sender = _msgSender();
    require(
        !_isExcluded[sender],
        "Excluded addresses cannot call this function"
    );
    (uint256 rAmount, , , , , , ) = _getValues(tAmount);
    _rOwned[sender] = _rOwned[sender].sub(rAmount);
    _rTotal = _rTotal.sub(rAmount);
    _tFeeTotal = _tFeeTotal.add(tAmount);
}
```

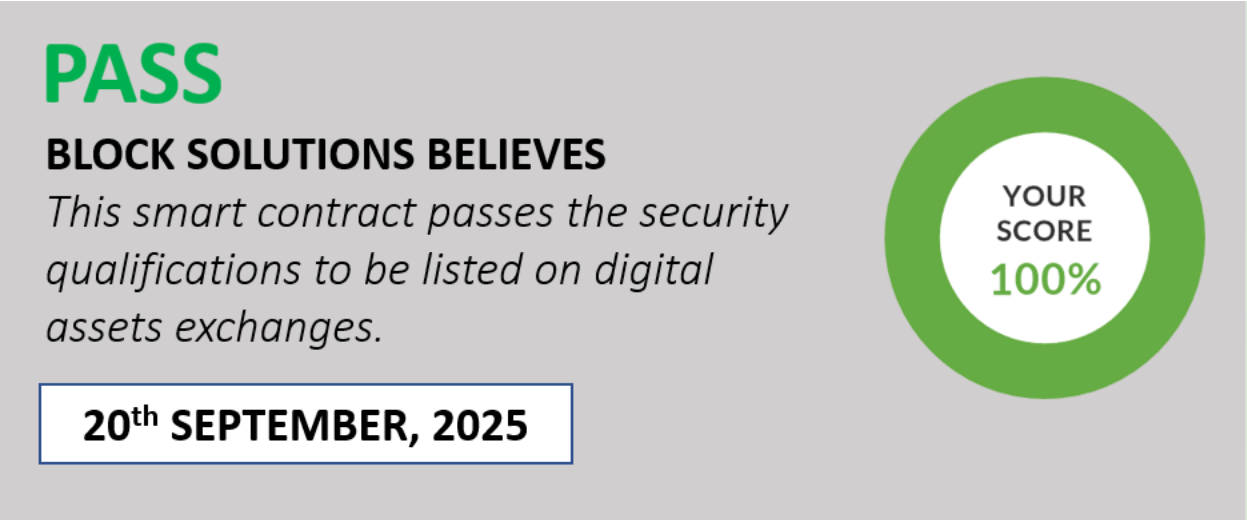
This will let the contract owner update the charity fee percentage, where charityFeeBps is given in basis points (e.g., 100 = 1%). The function updates _charityFee and then verifies that the combined total of all fees (_taxFee + _liquidityFee + _charityFee) does not exceed the maximum cap of 25% (MAX_FEE).

```
function setCharityFeePercent(uint256 charityFeeBps) external onlyOwner {
    _charityFee = charityFeeBps;
    require(
        _taxFee + _liquidityFee + _charityFee <= MAX_FEE,
        "Total fee is over 25%"
    );
}
```

This will let the contract owner set the threshold of tokens that must accumulate in the contract before triggering the automatic liquidity-adding process. The _amount must be at least 0.05% of the total supply, otherwise the transaction reverts. Once set, numTokensSellToAddToLiquidity is updated and an event SwapAndLiquifyAmountUpdated is emitted.

```
function setSwapBackSettings(uint256 _amount) external onlyOwner {
    require(
        _amount >= totalSupply().mul(5).div(10**4),
        "Swapback amount should be at least 0.05% of total supply"
    );
    numTokensSellToAddToLiquidity = _amount;
    emit SwapAndLiquifyAmountUpdated(_amount);
}
```

Testing Summary



Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert () misuse	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	**Passed**

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is **Well-Secured.** Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Quick Stat section.
We found 0 critical, 0 high, 0 medium and 0 low level issues.

Code Quality

The ALBUBUX Smart Contract protocol consists of one smart contract. It has other inherited contracts like IERC20, Ownable, BaseToken. These are compact and well written contracts. Libraries used in ALBUBUX Smart Contract are part of its logical algorithm. They are smart contracts which contain reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The BLOCKSOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.
Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a ALBUBUX Smart Contract smart contract code in the form of File.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on

well-known industry standard open-source projects. And even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No Critical severity vulnerabilities were found.

High

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Suggestions:

1. **Require service fee paid & use call**
   - Problem: constructor uses .transfer(serviceFee_) and may revert if serviceFee_ > msg.value or if receiver is a contract.
   - Suggestion: require(serviceFee_ <= msg.value) and replace .transfer with (bool sent,) = payable(...).call{value: serviceFee_}("") and check sent.

2. **Validate router/factory and handle existing pair**
   - Problem: arbitrary router address / createPair may revert or be malicious.
   - Suggestion: check factory != address(0), read existing pair with getPair(...) and call createPair only if none exists; consider try/catch for safety.

3. **Add owner controls/events & reversibility**
   - Problem: owner can exclude addresses and change fees but there are few events and exclusions are irreversible for fees.
   - Suggestion: add includeInFee(address), includeInReward(address), and emit events for ExcludeFromFee, IncludeInFee, TaxFeeUpdated, LiquidityFeeUpdated, CharityFeeUpdated, CharityAddressUpdated, SwapAndLiquifyToggled.

4. **Add timelock / multisig for critical admin actions**
   - Problem: owner can instantly change fees up to 25% or exclude users (centralization & rug risk).
   - Suggestion: require multisig or a timelock (or at least multi-sig) for fee/charity/major admin updates in production.

5. **Limit or rework _excluded scanning (gas DoS)**
   - Problem: _getCurrentSupply() loops over _excluded on every transfer — O(N) gas growth; owner could bloat list.
   - Suggestion: enforce a max excluded count, document cost, or maintain running rSupply/tSupply adjustments on exclude/include to avoid scanning each transfer.

6. **Expose a toggle and event for swapAndLiquifyEnabled**
   - Problem: currently set true in constructor with no setter.
   - Suggestion: add setSwapAndLiquifyEnabled(bool) (onlyOwner) and an event so you can pause automatic liquidity behavior if necessary.

7. **Rescue functions for ETH/ERC20**
   - Problem: no owner recovery for accidentally sent tokens/ETH.
   - Suggestion: add rescueERC20(token, to, amount) and rescueETH(to, amount) (onlyOwner) with events — but document policy (avoid rescuing user funds).

8. **Make charity handling explicit / optional auto-swap**

- Problem: charity receives tokens (not ETH); teams may expect ETH donations.
- Suggestion: add an opt-in swapCharityTokensToETH flow (carefully, reentrancy-guarded) or a helper distributeCharity() that owner/charity can call to convert tokens to ETH and forward funds.

9. **Add checks & documentation for decimals and minimum transfer rounding**
   - Problem: small transfers can round fees to zero due to integer division and decimals = 9.
   - Suggestion: document token unit conversions in README and add tests that show minimum amounts where fees apply.

10. **Require totalSupply_ > 0 and sanity checks in constructor**
    - Problem: no guard for zero total supply or invalid charity/router addresses.
    - Suggestion: add require(totalSupply_ > 0) and require(router_ != address(0)) and similar sanity checks.

11. **Consider making LP recipient configurable or document LP burn**
    - Problem: LP tokens are sent to 0xdead (permanently locked) — important design choice.
    - Suggestion: document this clearly or allow owner/setter to send LP to a multisig or governance address (if intended).

12. **Remove redundant SafeMath (optional) and optimize bytecode**
    - Problem: Solidity 0.8 has built-in overflow checks; SafeMath is redundant and inflates bytecode.
    - Suggestion: remove SafeMath usage for gas/size savings.

13. **Add thorough unit tests & fuzzing for these edge cases**
    - Tests: (a) transfers between excluded/non-excluded; (b) tiny amounts where fee rounds to 0; (c) repeated excludes to show gas growth; (d) swapAndLiquify trigger under various volumes; (e) constructor with existing pair and with malicious router stub.

14. **Add clear on-chain/off-chain audit notes & an operations playbook**
    - Problem: auditors and ops need to know how to react (e.g., emergency pause).
    - Suggestion: add a short on-chain changelog events + off-chain runbook (how to pause, rescue, change charity) and recommend renouncing ownership only after mechanisms are tested.

15. **Document centralization risk & recommend governance**
    - Problem: owner privileges are broad (fee changes, exclusions).
    - Suggestion: call out this risk in the report and recommend multisig/timelock and public disclosure of owner/charity keys.

Conclusion

The Smart Contract code passed the audit. We were given a contract code with some considerations to take. And we have used all possible tests based on given objects as files. Since possible test cases can be unlimited for such extensive smart contract protocol, hence we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in Quick Stat section of the report. Audit report contains all found security vulnerabilities and other issues in the reviewed code.
Security state of the reviewed contract is "Well Secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally, follow a process of first documenting the suspicion with unresolved questions, then

confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.